ECE 375 LAB 7

Lab 7 – Remotely communicated Rock Paper Scissors

Lab Time: Thursday 12-2

Author name: Noah Bean

Programming partner name: Daniel Lee

INTRODUCTION

The purpose of this lab was to write an assembly program for two separate atmega32u4 boards and have them interact, learn how to configure and use the Universal Synchronous/Asynchronous Receiver/Transmitter (USART) module on the ATmega32U4 microcontroller, and learn how to configure and use the 16-bit Timer/Counter1 module to generate a 1.5-sec delay. The objective of this lab was to get two ATmega32u4 boards to play a game of rock, paper, scissors. In order to fulfill this objective of lab 7, an AVR assembly program was devised for two ATmega32U4 boards to engage in a rock, paper, scissors game. This required configuring and utilizing the USART module for communication, and the Timer/Counter1 modules to create a 1.5-second delay. The process involved USART Configuration with Initializing USART for communication on both boards by setting the baud rate, frame format, and enabling transmitter and receiver. Timer/Counter1 Initialization required configuring Timer/Counter1 on both boards to generate the required delay, adjusting the prescaler values and compare match values, implementing button press detection for PD4 and PD7 for user hand choice selection and game start. Button 7 synchronized the boards by starting the game by displaying the game start message and calling the game start function, while button 4 cycled through rock, paper, and scissors options iteratively. Game flow control occurred after board synchronization, allowing each board to choose its move. Upon selection, transmitting the choice to the opponent board via USART and receiving the opponent's hand choice. The final score was evaluated by factoring in the user's hand and receiving the opponent's choice to decide the game's outcome, determining the winner, loser, or if it was a draw. The result was then displayed on the LCD on line 1. After displaying the resulting strings, the boards were reset to the initial welcome message and the initialization process was restarted and the next round could be started. The game loop continued forever unless the board was reset. This summary provides a broad perspective, with the actual implementation in the Design section. Testing and debugging results appear in the testing section.

DESIGN

The boards must be set up to communicate

Board 1 Board 2

PD2 <-> PD3

PD3 <-> PD2

PD.gnd <-> PD.gnd

The two boards also shared power

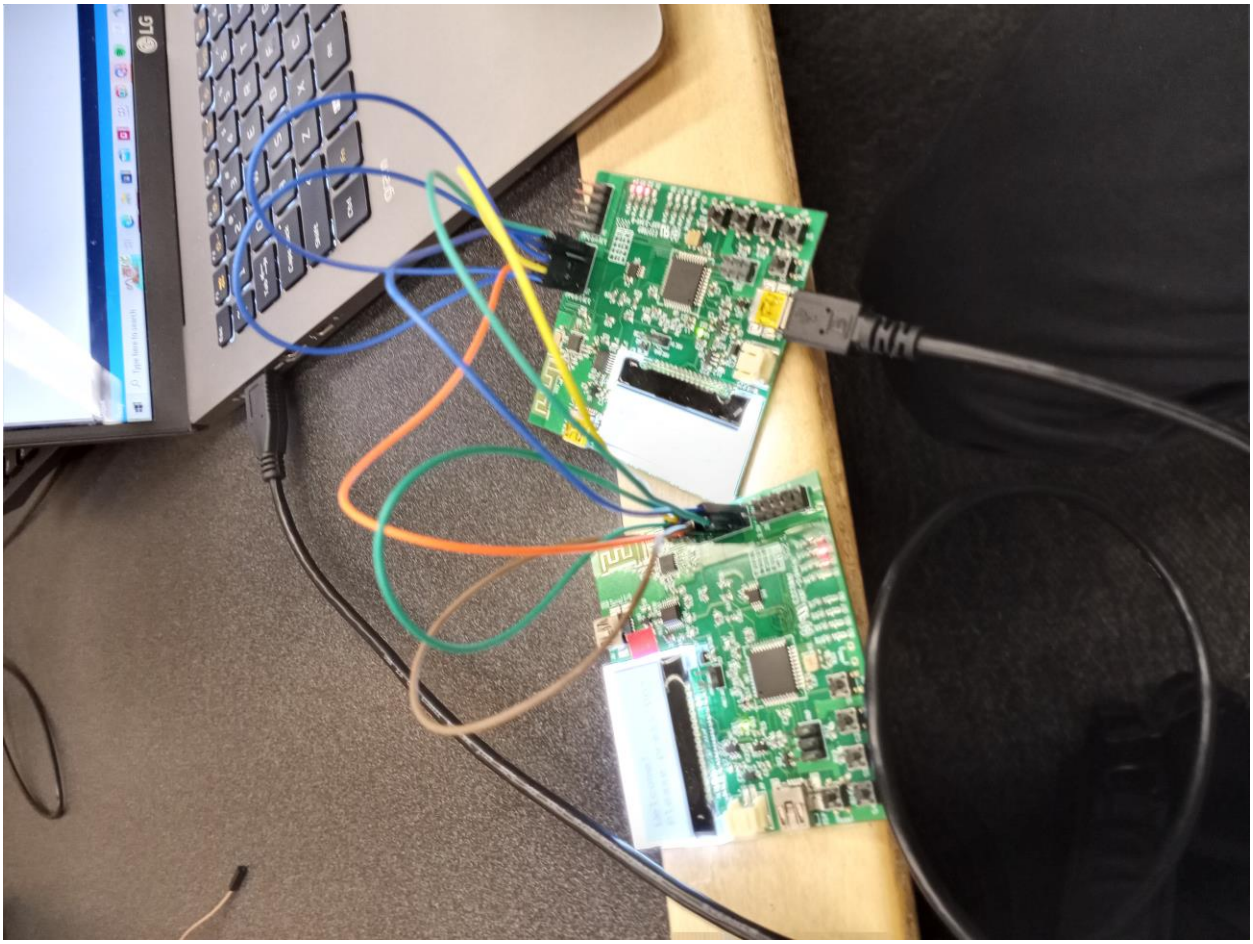

Fig 1: An image of the two boards properly set

INITIALIZATION ROUTINE

Stack pointer must point to the RAM memory at the end of the RAM

Set up timers

The Timer/Counter1 must be set for Normal mode by using WGM13:10 = 0000

The prescaler is 1024

LCD display is initialization with LCDINIT

Turn on the LCD display backlight with LCDBacklightOn

Clear random characters with LCDCLr

Set up I/O for B, D

Buttons PD7 and PD 4 are configured for input

The LEDs are initialized as output

Set up USART for transmitting and receiving data

the baud rate for USART1 is Set to 2400 bps.

The formula: UBRR = clock frequency/(8*baud rate)

Was used to calculate 8000000/(8*2400) = 416 = UBRR

USART1 is configured to receive and send data with 2 bits for stop and 8 bits for data

Set up interrupts for int0, int1, and $0032

The EIMSK register is set to enable some of the buttons, but button 7 and 4 should be disabled initially

INT0, INT1 trigger on falling

Global interrupts are enabled with SEI

MAIN ROUTINE

The main function orchestrates the flow of the game. Initially, it enables PD7 for signaling the display of the welcome string on the LCD. Subsequently, the start string is written to the LCD. Following this, a bitwise AND operation is executed between the user ready register and the opponent ready register to verify the readiness of both boards. Upon confirmation, the DisplayStart function and Full_Delay function are used, ensuring synchronization of the boards by monitoring the delay_done condition. interrupts get disabled, the opponents hand is written to the first line. A further delay is introduced using the Full_Delay function, after which the evaluate_score function is used to compare the user's and opponent's hands and determine the round's winner. After, the main function jumps to the initialization in order to reset the initialization values.

SUBROUTINES

1. GameStart

   This function operates by first displaying the "game ready" string. Then, it enters a loop where it continually displays lines on the LCD, increments the user_ready register, moves the user_ready register value into the SendData register, and calls the transmit function. Queued interrupts are cleared by resetting the EIFR register. The LCD gets cleared. The state of the mpr (memory protection register) and SREG (status register) are preserved by pushing them onto the stack and popping them afterward.

2. DisplayStart

   This function is used to write the "Game Start" message on the first line of the LCD. It works by getting the string that was hard coded into the Z and then writing it to the LCD using LCDWrite.

3. Choose_hand

   This function is associated with PD4 and alters the user's hand choice upon being pressed. It cycles through rock, paper, scissors, and back to rock. Initially, the function deals with debouncing. It sets the user's hand to 0 and then increments it. If the user's hand is rock -> paper, paper -> scissors, scissors -> rock. Finally, the user choice is sent to the opponent by way of the transmit function.

4. Transmit

   This function is responsible for transmitting data to the other board. Initially, it enters a loop called Transmit_Wait, where it waits until the USART Data Register Empty (UDRE1) flag is set, indicating that it's safe to write new data. Once UDRE1 is empty, the data in the SendData variable is transferred to the USART Data Register (UDR1), signaling to the other board that a message is ready to be sent.

5. Received

   This function gets data received from the other board and moves it to the correct register. Initially, it loads the data from the USART Data Register and compares it to 1 since this is the bitwise AND message that shows both boards are ready. If the data equals 1, the function places data into the opponent_is_ _ready register. otherwise, if the data is not 1, it is moved into the opponent's_hand_register. To preserve the register values, they are pushed onto the stack before the function and then popped after the function is completed. The function returns.

6. Evaluate_score

   This function determines whether the user has won, lost, or drawn in the game and calls the correct function to display the result to the first line of the LCD. Initially, it subtracts the opponent's hand from the user's hand and stores this value in the user hand register. Then, based on the result of this

subtraction which could be 0, -1, -2, -3, 1, 2, 3, it displays the appropriate message indicating whether the user has won, lost, or the game resulted in a draw on the LCD display. The function works by evaluating the result of subtracting with a table of hardcode numbers that correspond to rock beating scissors, scissors beating paper, and paper beating rock.

rock = 1

paper = 2

scissors = 4

1 - 1 = 0 tie

1 - 2 = -1 loss

1 - 4 = -3 win

2 - 1 = 1 win

2 - 2 = 0 tie

2 - 4 = -2 loss

4 - 1 = 3 loss

4 - 2 = 2 win

4 - 4 = 0 tie

7. display_opp_rock

   This function writes the opponent rock string to the LCD display line 1

8. display_opp_paper

   This function writes the opponent paper string to the LCD display line 1

9. display_opp_scissors:

   This function writes the opponent scissors string to the LCD display line 1

10. Full_delay

    This function implemented a delay of six seconds, during which each of 4 LED lights are turned off sequentially every 1.5 seconds. First, the timer/counter is initialized with a prescale value of 1024. Then, The LED_MASK is set to $F0, to indicate that 4 LEDs in are turned on. After, a loop is executed, to count up to 1.5 seconds, during which all LEDs are initially on. After 1.5 seconds 1 LED from left to right is turned off. This process is repeated until all four LEDs have been turned off. Once the delay_done flag is set, indicating that the delay is completed, the function returns. This mechanism ensures that each LED remains on for 1.5 seconds before being turned off so that a total of six seconds has been delayed , and each of the 4 lights has been sequentially turned off.

    Display_win

    This function writes the “you won” string to the LCD display line 1

11. Display_draw

    This function writes the “draw…” string to the LCD display line 1

12. Display_lost

    This function writes the “you lost” string to the LCD display line 1

13. display_opp_rock

14. This function writes the rock string to the LCD display line 1

15. display_rock

    This function writes the rock string to the LCD display line 2

16. display_opp_paper

    This function writes the paper string to the LCD display line 1

17. display_paper

    This function writes the paper string to the LCD display line 2

18. display_opp_scissors

    This function writes the scissors string to the LCD display line 1

19. display_scissors

    This function writes the scissors string to the LCD display line 2

20. Wait Routine

The Wait routine from previous labs was used for debugging. It works by filling a register and decrementing it until it is 0 and using this operation time to wait.

Stored Program Data

```
"Welcome!        Please press PD7"

"Ready. Waiting  for the opponent"

"Game start                      "

"Rock            "

"Paper           "

"Scissor         "

"You won!        "

"Draw...         "
```

```
Additional Program includes

 "LCDDriver.asm"
```

TESTING

Case	Expected	Actual meet expected
PD7 Pressed	“Welcome message“ displayed	Yes
PD7 Pressed Again	“Game Start message“ displayed	Yes
PD4 Pressed	“Rock“ is selected for user hand	Yes
PD4 Pressed again	“paper“ is selected for user hand	Yes
PD4 Pressed again	“scissors“ is selected for user hand	Yes
PD4 Pressed again	“Rock“ is selected for user hand	Yes
User plays rock, opponent plays rock	“draw message“ displayed	Yes
User plays rock, opponent plays paper	“lost message“ displayed	Yes
User plays rock, opponent plays scissors	“won message“ displayed	yes

User plays paper, opponent plays paper	“draw message“ displayed	Yes
User plays paper, opponent plays scissors	“lost message“ displayed	Yes
User plays paper, opponent plays rock	“won“ message displayed	yes
User plays scissors, opponent plays scissors	“draw message“ displayed	Yes
User plays scissors, opponent plays rock	“lost message” displayed	Yes
User plays scissors, opponent plays paper	“won message“ displayed	yes

STUDY QUESTIONS

There were no study questions for lab 7

DIFFICULTIES

Difficulties for this lab include testing the code, creating a control flow algorithm, and integrating code.

CONCLUSION

The purpose of this lab was to write an assembly program for two separate atmega32u4 boards and have them interact, learn how to configure and use the Universal Synchronous/Asynchronous Receiver/Transmitter (USART) module on the ATmega32U4 microcontroller, and learn how to configure and use the 16-bit Timer/Counter1 module to generate a 1.5-sec delay. The objective of this lab was to get two ATmega32u4 boards to play a game of rock, paper, scissor. In order to fulfill this objective of lab 7, an AVR assembly program was devised for two ATmega32U4 boards to engage in a rock, paper, scissors game. This required configuring and utilizing the USART module for communication, and the Timer/Counter1 modules to create a 1.5-second delay. The process involved USART Configuration with Initializing USART for communication on both boards by setting the baud rate, frame format, and enabling transmitter and receiver. Timer/Counter1 Initialization required configuring Timer/Counter1 on both boards to generate the required delay, adjusting the prescaler values and compare match values, implementing button press detection for PD4 and PD7 for user hand choice selection and game start. Button 7 synchronized the boards by starting the game by displaying the game start message and calling the game start function, while button 4 cycled through rock, paper, and scissors options iteratively. Game flow control occurred after board synchronization, allowing each board to choose its move. Upon selection, transmitting the choice to the opponent board via USART and receiving the opponent’s hand choice. The final score was evaluated by factoring in the user’s hand and receiving the opponent's choice to decide the game's outcome, determining the winner, loser, or if it was a draw. The result was then displayed on the LCD on line 1. After displaying the resulting strings, the boards were reset to the initial welcome message and the initialization process was restarted and the next round could be started. The game loop continued forever unless the board was reset. This summary

provides a broad perspective, with the actual implementation in the Design section. Testing and debugging results appear in the testing section.

SOURCE CODE

```
;***********************************************************
;*
;*     This is the TRANSMIT skeleton file for Lab 7 of ECE 375
;*
;*     Rock Paper Scissors
;*     Requirement:
;*     1. USART1 communication
;*     2. Timer/counter1 Normal mode to create a 1.5-sec delay
;***********************************************************
;*
;*      Author: Daniel Lee and Noah Bean
;*        Date: 3/17/2024
;*
;***********************************************************

.include "m32U4def.inc"         ; Include definition file


;TO-DO:
; If receivedata == senddata, then we can go start the game
;
;

;***********************************************************
;*  Internal Register Definitions and Constants
;***********************************************************
.def    mpr = r16               ; Multi-Purpose Register

.def   opponent_hand = r17               ;for opponent hand

;*
;*     WARNING: Register r20-r22 are reserved and cannot be
;*              renamed outside of the LCD Driver functions. Doing
;*              so will damage the functionality of the LCD Driver

.def   user_hand = r23                          ;to see how hand is set
.def   SendData = r24
.def   ReceivedData = r25
.def   one_half_cnt = r18          ;for delay
.def   user_ready = r10            ;to tell the opponent that user is ready
.def   opp_ready = r9                     ;to tell the user that opponent is ready
.def   delay_done = r19            ;flag to see if delay is over
.def LED_MASK = r11    ; Mask for PORTB 7:4 LEDs

.equ   change_hand = 4 ;pb4
.equ   ready_signal = 7 ;pb7

;*     The upper 16 characters should be located in SRAM starting at 0x0100.
;*     The lower 16 characters should be located in SRAM starting at 0x0110.
; -> L1 = 00, H1 = 01, L2 = 10, H2 = 01
.equ   L1 = $00             ; LCD String 1 low
.equ   H1 = $01             ; LCD String 1 high
```

```
.equ   L2 = $10             ; LCD String 2 low
.equ   H2 = $01             ; LCD String 2 high

; Use this signal code between two boards for their game ready
.equ    SendReady = 0b11111111
.equ Rock = 1 ;for evaluating score
.equ Paper = 2 ;for evaluating score
.equ Scissors = 4 ;for evaluating score

.equ ChooseRock = $01 ;for choosing hand
.equ ChoosePaper = $02 ;for choosing hand
.equ ChooseScissors = $03 ;for choosing hand

;***********************************************************
;*  Start of Code Segment
;***********************************************************
.cseg                           ; Beginning of code segment

;***********************************************************
;*  Interrupt Vectors
;***********************************************************
.org    $0000                   ; Beginning of IVs
          rjmp    INIT                  ; Reset interrupt

.org   $0002;For push button 4 (PIND0->PB4)
              rcall choose_hand                 ;interrupt to cycle through rock paper
scissors
              reti

.org   $0004;For push button 7 (PIND1->PB7)
              rcall GameStart                   ;interrupt to start the game
              reti

;.org  $0022
;             rcall Full_Delay
;             reti

              ;$0006 and $0008 occupied for transmitter and receiver
.org   $0032
              rcall Received                    ;interrupt that indicates that signal is
received
              reti

.org    $0056                   ; End of Interrupt Vectors

;***********************************************************
;*  Program Initialization
;***********************************************************
INIT:
              ;Stack Pointer (VERY IMPORTANT!!!!)
              ldi mpr, low(RAMEND)
              out SPL, mpr
              ldi mpr, high(RAMEND)
              out SPH, mpr

              ;cli

              ;clr delay_done
```

```
		;clr user_ready
		;clr opp_ready
		;clr user_hand
		;clr opponent_hand
		;clr ReceivedData
		;clr SendData
		;ldi user_hand, $00
		;I/O Ports
		ldi mpr, 0b11110000
		out DDRB, mpr
		ldi mpr, 0b00000000
		out PORTB, mpr
		ldi mpr, 0b00001000
		out DDRD, mpr
		ldi mpr, 0b11111011; PD7 and PD4 are what matter, but we can set all
(active high) initially
		out PORTD, mpr

		rcall LCDInit
		rcall LCDCLr
		rcall LCDBacklightOn


	;USART1
		;Set baudrate at 2400bps (double rate) UBRR = clock frequency/(8*baud rate)
		ldi mpr, high(416);
		sts UBRR1H, mpr;
		ldi mpr, low(416); 8000000/(8*2400) = 416 = UBRR
		sts UBRR1L, mpr;
		;Enable receiver and transmitter
		;Set frame format: 8 data bits, 2 stop bits
		ldi mpr, 0b00100010; data register empty = 1; double rate = 1;
		sts UCSR1A, mpr
		ldi mpr, 0b10011000; receiver complete enable = 1; receiver enable,
transmitter enable = 1;
		sts UCSR1B, mpr
		ldi mpr, 0b00001110; UCPOL1 = 00, USBS1 = 1, UCSZ = 011 (only last two bits
relevant for UCSR1C), UMSEL1 = 00, UPM1 = 00
		sts UCSR1C, mpr; in extended I/O space, so we use sts


	;TIMER/COUNTER1
		;Set Normal mode (WGM 13:10 = 0000)
		ldi mpr, 0b00000000; no non-invert or invert since normal mode
		sts TCCR1A, mpr
		ldi mpr, 0b00000101; clock selection 1024 prescale, so 101 (we want to get
a big delay. if wrong, it can be EDITTED)
		sts TCCR1B, mpr

;		ldi mpr, 0b00000000			;disable counter
;		sts TIMSK1, mpr

		;EICRA and EIMSK
		ldi mpr, 0b00001010; using INT0 and INT1 (falling edge)
		sts EICRA, mpr
		ldi mpr, 0b00000000
		out EIMSK, mpr
```

```
            sei; Enable global interrupt
;***********************************************************
;*  Main Program
;***********************************************************
MAIN:

            ;cpi delay_done, $01
            ;breq Evaluate_Result
            clr delay_done
            clr user_ready
            clr opp_ready
            clr user_hand
            clr opponent_hand
            clr ReceivedData
            clr SendData
            ldi           mpr,$FF             ;to prevent queued interrupts
            out           EIFR, mpr

            ;welcome message
            ldi ZL, LOW(STRING_START<<1)
            ldi    ZH, HIGH(STRING_START<<1)
            ldi YL, $00
            ldi YH, $01 ;$0100 first bit of LCD

            LOOP_START:
                  lpm mpr, Z+
                  st Y+, mpr
                  tst mpr
                  breq LOOP_START_DONE
                  rjmp LOOP_START

LOOP_START_DONE:

            rcall LCDWrite

            ldi mpr, 0b00000010         ;allow the users to press ready button
            out EIMSK, mpr
            mov  mpr, opp_ready   ; Move the value of user_ready to register mpr
            and  mpr, user_ready    ; Perform a bitwise AND operation between
user_ready and opp_ready

            cpi    mpr, $01
            breq Play_Game                           ;both players ready, start the
game
            rjmp LOOP_START_DONE

      Play_Game:                                ;the second LCD line is being handled by
interrupt (pd4 (INT0))
            ;rcall LCDBacklightOff
            clr user_ready
            clr opp_ready
            ldi mpr, 0b00000001         ;allow the users to press change_hand button
            out EIMSK, mpr
            rcall DisplayStart          ;this function displays "Game Start" on first
line and enables change_hand button
            rcall Full_Delay
            cpi delay_done, $01
            breq Display_Both
```

```
		;rcall LCDBacklightOff
		rjmp Play_Game

	Display_Both: ;we display the opponent's data onto the first line while we display
our own data onto the second line for six seconds
		;clr delay_done	;clear it so it only sets when next delay is actually
over
		;rcall LCDBacklightOff
		; disable PD4
		ldi mpr, 0b00000000
		out EIMSK, mpr

		;rcall Full_Delay

		cpi opponent_hand, Rock		;just all comparisons for opponent hand
		breq Opp_Rock			;

		cpi opponent_hand, Paper	;
		breq Opp_Paper				;

		cpi opponent_hand, Scissors;
		breq Opp_Scissors		;
		rjmp Display_Both		;

	Opp_Rock:
		rcall display_opp_rock		;
		rjmp Hands_Delay		;
	Opp_Paper:					;
		rcall display_opp_paper		;
		rjmp Hands_Delay		;
	Opp_Scissors:				;
		rcall display_opp_scissors	;
		rjmp Hands_Delay		;


	Hands_Delay:				;
		rcall LCDBacklightOn	;
		rcall Full_Delay	;might want to place this right after Display
Both:
		cpi delay_done, $01	;
		breq Evaluate_Result ;
		;rjmp Hands_Delay
		rcall LCDClrLn1			;

	Evaluate_Result:		;

		mov	opponent_hand, ReceivedData ;this is because we use opponent_hand to
evaluate score
		rcall evaluate_score		;
		rcall Full_Delay			;
		rcall LCDClr				;

	rjmp MAIN

;***********************************************************
;*	Functions and Subroutines
;***********************************************************
;-----------------------------------------------------------
```

```
; Func: DisplayStart
; Desc: Cut and paste this and fill in the info at the
;             beginning of your functions
;-----------------------------------------------------------
DisplayStart:                                       ; Begin a function with a label

              ; Save variable by pushing them to the stack
              push   mpr                                        ; save mpr
              in            mpr, SREG                           ; save program state
              push   mpr

              ldi mpr, 0b00000001  ;enable change_hand button through interrupt
              out EIMSK, mpr
              rcall LCDClr
              ;rcall Full_Delay
              ; Execute the function here
              ldi ZL, LOW(STRING_PLAY<<1)
              ldi    ZH, HIGH(STRING_PLAY<<1)
              ldi YL, $00
              ldi YH, $01 ;$0100 first bit of LCD

              LOOP_PLAY:
                     lpm mpr, Z+
                     st Y+, mpr
                     cpi ZL,  LOW(STRING_PLAY_WAIT<<1)
                     brne LOOP_PLAY

LOOP_PLAY_DONE:
              rcall LCDWrLn1

               ;Restore variable by popping them from the stack in reverse order
              pop           mpr
              out           SREG, mpr
              pop           mpr

              ret                                       ; End a function with RET

;-----------------------------------------------------------
; Func: ChooseHand
; Desc:
;The second line of the LCD is where the user is able to select the choice among
;three gestures (i.e., rock, paper, and scissors) that they want to send.
;• First pressing PD4 selects the Rock gesture.
;• Pressing PD4 changes the current gesture and iterates through the thee
;gestures in order. For example, Rock Ñ Paper Ñ Scissor Ñ Rock Ñ ...
;• Pressing PD4 immediately displays the choice on the user’s LCD.
;(Challenge part should meet all of the above features as well.)
;For example, if the display currently shows:
;Game start
;
;-----------------------------------------------------------
choose_hand:
; Save variable by pushing them to the stack
              push   mpr                                        ; save mpr
              in            mpr, SREG                           ; save program state
              push   mpr

              ;ldi          mpr,$FF              ;to prevent queued interrupts
```

```
		;out		EIFR, mpr

		cpi		user_hand, $03
		breq	reset_hand
		rjmp	no_reset

reset_hand:
		ldi		user_hand, $00; this is to reset when scissors and needs to
reset to 0 to properly cycle

no_reset:
		inc		user_hand;each time pd4 is pressed, we must increment

cycle_hand:
	cpi		user_hand, $01 ;$01
	breq	rock_choice_tag

	cpi		user_hand, $02 ;$02
	breq	paper_choice_tag

	cpi		user_hand, $03 ;$03
	breq	scissor_choice_tag
	;rjmp	cycle_hand

	rock_choice_tag:
		ldi		user_hand, rock	; iterate choice
		rcall	display_rock	; write rock to LCD
		;in mpr, PORTB
		;cpi delay_done, $01
		rjmp choose_hand_end_tag
		;rjmp cycle_hand

	paper_choice_tag:
		ldi		user_hand, paper	; iterate choice
		rcall	display_paper		; write paper to LCD
		;in mpr, PORTB
		;cpi delay_done, $01
		rjmp choose_hand_end_tag
		;rjmp	cycle_hand

	scissor_choice_tag:
		ldi		user_hand, scissors ; iterate choice
		rcall	display_scissors	; write scissors to LCD
		;in mpr, PORTB
		;cpi delay_done, $01
		rjmp choose_hand_end_tag

	choose_hand_end_tag:
		;ldi mpr, 0b00000000
		;out EIMSK, mpr		;disable pd4
		mov	SendData, user_hand	;whatever user_hand holds at the end of the
delay is what we transmit
		rcall Transmit

		;ldi mpr, 0b00000001
		;out EIFR, mpr
		; Restore variable by popping them from the stack in reverse order
		pop		mpr
```

```
                out             SREG, mpr
                pop             mpr

                ret                                             ; End a function with RET
;-----------------------------------------------------------
; Func: GameStart
; Desc: Cut and paste this and fill in the info at the
;               beginning of your functions
;-----------------------------------------------------------
GameStart:                                                      ; Begin a function with a label

                ; Save variable by pushing them to the stack
                push    mpr                                             ; save mpr
                in              mpr, SREG                               ; save program state
                push    mpr

                ldi             mpr,$FF                 ;to prevent queued interrupts
                out             EIFR, mpr

                rcall LCDClr
                ; Execute the function here
                ldi ZL, LOW(STRING_READY<<1)
                ldi     ZH, HIGH(STRING_READY<<1)
                ldi YL, $00
                ldi YH, $01 ;$0100 first bit of LCD

                LOOP_THROUGH_READY:
                        lpm mpr, Z+
                        st Y+, mpr
                        cpi ZL,  LOW(STRING_READY_WAIT<<1)
                        brne LOOP_THROUGH_READY

LOOP_READY:
                rcall LCDWrite
                inc user_ready
                mov SendData, user_ready
                rcall Transmit                  ;This sends the signal that you are ready and
triggers the interrupt $0032 (I believe)

                ;ldi mpr, 0b00000010
                ;out EIFR, mpr

                ; Restore variable by popping them from the stack in reverse order
                pop             mpr
                out             SREG, mpr
                pop             mpr

                ret                                             ; End a function with RET

;-----------------------------------------------------------
; Func: Transmit
; Desc: Cut and paste this and fill in the info at the
;               beginning of your functions
;-----------------------------------------------------------
Transmit:                                                       ; Begin a function with a label

                push    mpr                                             ; save mpr
                in              mpr, SREG                               ; save program state
```

```
              push    mpr
Transmit_Wait:
              lds           mpr, UCSR1A
              sbrs    mpr, UDRE1           ; wait until UDRE1 is empty and is set to sts
UDR1,mpr
              rjmp    Transmit_Wait

              ;mov    mpr, SendData        ;signal that let's other board know it's ready
              sts UDR1, SendData

              ; Restore variable by popping them from the stack in reverse order
              pop           mpr
              out           SREG, mpr
              pop           mpr

              ret                                        ; End a function with RET

;-----------------------------------------------------------
; Func: Received
; Desc: Cut and paste this and fill in the info at the
;             beginning of your functions
;-----------------------------------------------------------
Received:                                                ; Begin a function with a label

              push    mpr                                        ; save mpr
              in            mpr, SREG                            ; save program state
              push    mpr

              ;lds ReceivedData, UDR1            ;UDR1 should be $FF if received SendReady
signal from other device
              lds ReceivedData, UDR1
              ;cpi    ReceivedData, SendReady
              ;breq Move_Opp_Ready
              ;rjmp Move_Opp_Hand
;Move_Opp_Ready:
              mov     opp_ready, ReceivedData
;             rjmp Receive_Done
;Move_Opp_Hand:
              mov     opponent_hand, ReceivedData
;Receive_Done:

              ; Restore variable by popping them from the stack in reverse order
              pop           mpr
              out           SREG, mpr
              pop           mpr

              ret                                        ; End a function with RET

;-----------------------------------------------------------
; Func: evaluate_score
; Desc: Checks which board has winning condition
;-----------------------------------------------------------
evaluate_score:                                          ; Begin a function with a
label

              ; Save variable by pushing them to the stack
              push    mpr                                        ; save mpr
              in            mpr, SREG                            ; save program state
```

```
                push   mpr

                ; Execute the function here
                /*
                rock = 1
                paper = 2
                scissors = 4

                1 - 1 = 0 tie
                1 - 2 = -1 loss
                1 - 4 = -3 win

                2 - 1 = 1 win
                2 - 2 = 0 tie
                2 - 4 = -2 loss

                4 - 1 = 3 loss
                4 - 2 = 2 win
                4 - 4 = 0 tie
                */

;sub this board from other board
                sub user_hand, opponent_hand
                rjmp check_draw

display_draw_tag:
                rcall display_draw
                rjmp end_evaluate

display_win_tag:
                rcall display_win
                rjmp end_evaluate

display_lost_tag:
                rcall display_lost
                rjmp end_evaluate


check_draw:
                ;compare if difference is 0
                cpi user_hand, 0
                ; if not, skip to next tag
                breq display_draw_tag
                rjmp check_win
                ;brne check_win
                ; else, load tie string to evaluate register


check_win:
                ;compare if difference is -3, 1, 2
                cpi user_hand, -3
                breq display_win_tag

                cpi user_hand, 1
                breq display_win_tag

                cpi user_hand, 2
                breq display_win_tag
```

```
		rjmp check_lose


check_lose:
		;compare if difference is 3, -1, -2
		cpi user_hand, 3
		breq display_lost_tag

		cpi user_hand, -1
		breq display_lost_tag

		cpi user_hand, -2
		breq display_lost_tag

		;rjmp end_evaluate

end_evaluate:

		; Restore variable by popping them from the stack in reverse order
		pop		mpr
		out		SREG, mpr
		pop		mpr
		ret					; End a function with RET

;***********************************************************
; Sub: display_win
; Desc:	A function to display win message if won
;***********************************************************
display_win:

		push	mpr				; save mpr
		in		mpr, SREG			; save program state
		push	mpr

		rcall LCDClrLn1
		ldi		ZL, LOW(STRING_WON<<1)	; Z <- program memory address of
first character
		ldi		ZH, HIGH(STRING_WON_END<<1); ZL = , ZH =

		ldi		YL, L1				; Y <- data memory address
of character destination
		ldi		YH, H1				; YL = $00, YH = $01

		display_win_loop:                                           ;
			lpm mpr, Z+				; do { mpr <-
ProgramMemory[Z], Z++,
			st  Y+,  mpr			; DataMemory[Y] <- mpr, Y++ }
			cpi ZL,  LOW(STRING_WON_END<<1); while (Z != program memory address
after last character)
			brne display_win_loop                                   ;

	rcall		LCDWrLn1	; Writes to line 1 of LCD

	pop		mpr
		out		SREG, mpr
		pop		mpr
```

```
ret

;-----------------------------------------------------------
; Sub: display_draw
; Desc:       A function to display draw message if draw
;-----------------------------------------------------------
display_draw:

              push   mpr                                    ; save mpr
              in            mpr, SREG                       ; save program state
              push   mpr

              rcall LCDClrLn1
              ldi           ZL, LOW(STRING_DRAW<<1)     ; Z <- program memory address of
first character
              ldi           ZH, HIGH(STRING_DRAW_END<<1); ZL = , ZH =

              ldi           YL, L1                          ; Y <- data memory address
of character destination
              ldi           YH, H1                          ; YL = $00, YH = $01

              display_draw_loop:                                                    ;
                    lpm mpr, Z+                             ; do { mpr <-
ProgramMemory[Z], Z++,
                    st  Y+,  mpr                      ; DataMemory[Y] <- mpr, Y++ }
                    cpi ZL,  LOW(STRING_DRAW_END<<1); while (Z != program memory address
after last character)
                    brne display_draw_loop                                          ;

       rcall        LCDWrLn1      ; Writes to line 1 of LCD

       pop          mpr
              out           SREG, mpr
              pop           mpr

ret

;-----------------------------------------------------------
; Sub: display_lost
; Desc:       A function to display lost message if lost
;-----------------------------------------------------------
display_lost:

              push   mpr                                    ; save mpr
              in            mpr, SREG                       ; save program state
              push   mpr

              rcall LCDClrLn1
              ldi           ZL, LOW(STRING_LOST<<1)     ; Z <- program memory address of
first character
              ldi           ZH, HIGH(STRING_LOST_END<<1); ZL = , ZH =

              ldi           YL, L1                          ; Y <- data memory address
of character destination
              ldi           YH, H1                          ; YL = $00, YH = $01
              display_lost_loop:                                                    ;
                    lpm mpr, Z+                             ; do { mpr <-
ProgramMemory[Z], Z++,
```

```
                    st  Y+,  mpr                        ; DataMemory[Y] <- mpr, Y++ }
                    cpi ZL,  LOW(STRING_LOST_END<<1); while (Z != program memory address
after last character)
                    brne display_lost_loop                                          ;

       rcall        LCDWrLn1      ; Writes to line 1 of LCD

       pop          mpr
              out           SREG, mpr
              pop           mpr

ret

;-----------------------------------------------------------------
; Sub: display_rock
; Desc:       A function to display rock when needed
;-----------------------------------------------------------------
display_rock:

              push   mpr                                    ; save mpr
              in            mpr, SREG                       ; save program state
              push   mpr

              ;ldi user_hand, Rock
              rcall LCDClrLn2
              ldi           ZL, LOW(STRING_ROCK<<1)     ; Z <- program memory address of
first character
              ldi           ZH, HIGH(STRING_ROCK_END<<1); ZL = , ZH =

              ldi           YL, L2                          ; Y <- data memory address
of character destination
              ldi           YH, H2                          ; YL = $10, YH = $01
              display_rock_loop:                                          ;
                    lpm mpr, Z+                             ; do { mpr <-
ProgramMemory[Z], Z++,
                    st  Y+,  mpr                        ; DataMemory[Y] <- mpr, Y++ }
                    cpi ZL,  LOW(STRING_ROCK_END<<1); while (Z != program memory address
after last character)
                    brne display_rock_loop                                          ;

       rcall        LCDWrLn2      ; Writes to line 2 of LCD

              pop           mpr
              out           SREG, mpr
              pop           mpr

ret

;-----------------------------------------------------------------
; Sub: display_opp_rock
; Desc:       A function to display opponent rock when needed
;-----------------------------------------------------------------
display_opp_rock:

              push   mpr                                    ; save mpr
              in            mpr, SREG                       ; save program state
              push   mpr
```

```
		;ldi user_hand, Rock
		rcall LCDClrLn1
		ldi		ZL, LOW(STRING_ROCK<<1)	; Z <- program memory address of
first character
		ldi		ZH, HIGH(STRING_ROCK_END<<1); ZL = , ZH =

		ldi		YL, L1				; Y <- data memory address
of character destination
		ldi		YH, H1				; YL = $10, YH = $01
		display_opp_rock_loop:					;
			lpm mpr, Z+			; do { mpr <-
ProgramMemory[Z], Z++,
			st  Y+,  mpr		; DataMemory[Y] <- mpr, Y++ }
			cpi ZL,  LOW(STRING_ROCK_END<<1); while (Z != program memory address
after last character)
			brne display_opp_rock_loop				;

	rcall		LCDWrLn1	; Writes to line 1 of LCD

		pop		mpr
		out		SREG, mpr
		pop		mpr

ret

;-----------------------------------------------------------------
; Sub: display_paper
; Desc:	A function to display paper when needed
;-----------------------------------------------------------------
display_paper:

		push	mpr				; save mpr
		in		mpr, SREG			; save program state
		push	mpr

		;ldi user_hand, Paper
		rcall LCDClrLn2
		ldi		ZL, LOW(STRING_PAPER<<1)	; Z <- program memory address of
first character
		ldi		ZH, HIGH(STRING_PAPER_END<<1); ZL = , ZH =

		ldi		YL, L2				; Y <- data memory address
of character destination
		ldi		YH, H2				; YL = $10, YH = $01
		display_paper_loop:					;
			lpm mpr, Z+			; do { mpr <-
ProgramMemory[Z], Z++,
			st  Y+,  mpr		; DataMemory[Y] <- mpr, Y++ }
			cpi ZL,  LOW(STRING_PAPER_END<<1); while (Z != program memory
address after last character)
			brne display_paper_loop				;

	rcall		LCDWrLn2	; Writes to line 2 of LCD

		pop		mpr
		out		SREG, mpr
		pop		mpr
```

```
ret

;-----------------------------------------------------------
; Sub: display_opp_paper
; Desc:       A function to display opponent paper when needed
;-----------------------------------------------------------
display_opp_paper:

              push   mpr                                    ; save mpr
              in            mpr, SREG                       ; save program state
              push   mpr

              ;ldi user_hand, Paper
              rcall LCDClrLn1
              ldi           ZL, LOW(STRING_PAPER<<1)    ; Z <- program memory address of
first character
              ldi           ZH, HIGH(STRING_PAPER_END<<1); ZL = , ZH =

              ldi           YL, L1                          ; Y <- data memory address
of character destination
              ldi           YH, H1                          ; YL = $00, YH = $01
              display_opp_paper_loop:                                               ;
                    lpm mpr, Z+                             ; do { mpr <-
ProgramMemory[Z], Z++,
                    st  Y+,  mpr                      ; DataMemory[Y] <- mpr, Y++ }
                    cpi ZL,  LOW(STRING_PAPER_END<<1); while (Z != program memory
address after last character)
                    brne display_opp_paper_loop                                     ;

       rcall        LCDWrLn1      ; Writes to line 1 of LCD

              pop           mpr
              out           SREG, mpr
              pop           mpr

ret

;-----------------------------------------------------------
; Sub: display_scissors
; Desc:       Function to display scissors when needed
;-----------------------------------------------------------
display_scissors:

              push   mpr                                    ; save mpr
              in            mpr, SREG                       ; save program state
              push   mpr

              ;ldi user_hand, Scissors
              rcall LCDClrLn2
              ldi           ZL, LOW(STRING_SCISSOR<<1)  ; Z <- program memory address of
first character
              ldi           ZH, HIGH(STRING_SCISSOR_END<<1); ZL = , ZH =

              ldi           YL, L2                          ; Y <- data memory address
of character destination
              ldi           YH, H2                          ; YL = $10, YH = $01
              display_scissors_loop:                                                ;
```

```
                    lpm mpr, Z+                             ; do { mpr <-
ProgramMemory[Z], Z++,
                    st  Y+,  mpr                        ; DataMemory[Y] <- mpr, Y++ }
                    cpi ZL,  LOW(STRING_SCISSOR_END<<1); while (Z != program memory
address after last character)
                    brne display_scissors_loop                                  ;

      rcall         LCDWrLn2      ; Writes to line 2 of LCD

            pop           mpr
            out           SREG, mpr
            pop           mpr

ret

;-----------------------------------------------------------------
; Sub: display_opp_scissors
; Desc:       Function to display opponent scissors when needed
;-----------------------------------------------------------------
display_opp_scissors:

            push   mpr                                      ; save mpr
            in            mpr, SREG                         ; save program state
            push   mpr

            ;ldi user_hand, Scissors
            rcall LCDClrLn1
            ldi           ZL, LOW(STRING_SCISSOR<<1)  ; Z <- program memory address of
first character
            ldi           ZH, HIGH(STRING_SCISSOR_END<<1); ZL = , ZH =

            ldi           YL, L1                            ; Y <- data memory address
of character destination
            ldi           YH, H1                            ; YL = $00, YH = $01
            display_opp_scissors_loop:                                          ;
                    lpm mpr, Z+                             ; do { mpr <-
ProgramMemory[Z], Z++,
                    st  Y+,  mpr                        ; DataMemory[Y] <- mpr, Y++ }
                    cpi ZL,  LOW(STRING_SCISSOR_END<<1); while (Z != program memory
address after last character)
                    brne display_opp_scissors_loop
      ;

      rcall         LCDWrLn1      ; Writes to line 2 of LCD

            pop           mpr
            out           SREG, mpr
            pop           mpr

ret

;-----------------------------------------------------------------
; Sub: Full_Delay
; Desc:       A full delay for six seconds that turns off each LED light every 1.5
seconds.
;-----------------------------------------------------------------
Full_Delay:
```

```
        push    mpr                                     ; save mpr
        in              mpr, SREG                       ; save program state
        push    mpr

        ldi delay_done, $00         ;clear flag
    ;initialize timer/Counter1
    ;ldi mpr, (1 << CS12)|(1 << CS10)    ; Prescaler 1024, start timer
    ;sts TCCR1B, mpr
        ldi mpr, $F0
        mov LED_MASK, mpr
    ; Initialize delay counters
    ldi one_half_cnt, 4     ; Initialize counter for 1.5 second delay

tcount_loop:
    ; Load TCNT1 with initial value for 1.5 seconds delay
        ; clock cycles = 1.5 * 8,000,000 / 1024 = 11719
        ;65535(max) - 11719 = 53816 = $D238
    ldi mpr, $D2
    sts TCNT1H, mpr
    ldi mpr, $38
    sts TCNT1L, mpr

one_half_loop:
    ;Check if TCNT1 reached overflow (1.5 seconds elapsed)
    in mpr, TIFR1
    sbrc mpr, TOV1    ;skip if overflow flag is not set
    rjmp one_half_delay_done

    ; Delay loop
    rjmp one_half_loop

one_half_delay_done:
    ; Clear Timer1 overflow flag
    ldi mpr, (1 << TOV1)
    out TIFR1, mpr

    ;shift LED_MASK right to turn off one LED
    lsr LED_MASK

    ;LED_MASK to PORTB
    out PORTB, LED_MASK

    ;dec delay counter
    dec one_half_cnt
    brne tcount_loop    ; Repeat delay if counter not zero

        ldi delay_done, $01
    ;Turn off all LEDs at the end (just to make sure it works properly)
    ;andi mpr, $FF
    ;out PORTB, mpr
        pop             mpr
                out             SREG, mpr
                pop             mpr

    ret
;***********************************************************
;*     Stored Program Data
;***********************************************************
```

```
;-----------------------------------------------------------
; An example of storing a string. Note the labels before and
; after the .DB directive; these can help to access the data
;-----------------------------------------------------------
STRING_START:
    .DB             "Welcome!        Please press PD7"       ; Declaring data in ProgMem
(Use Z later to pull it out)
STRING_END:

STRING_READY:
.db "Ready. Waiting  for the opponent";DO NOT EDIT
STRING_READY_WAIT:

STRING_PLAY:
.db "Game start                     "
STRING_PLAY_WAIT:

STRING_ROCK:
.db "Rock            "
STRING_ROCK_END:

STRING_PAPER:
.db "Paper           "
STRING_PAPER_END:

STRING_SCISSOR:
.db "Scissor         "
STRING_SCISSOR_END:

STRING_LOST:
.db "You lost...     "
STRING_LOST_END:

STRING_WON:
.db "You won!        "
STRING_WON_END:

STRING_DRAW:
.db "Draw...         "
STRING_DRAW_END:

;***********************************************************
;*     Additional Program Includes
;***********************************************************
.include "LCDDriver.asm"          ; Include the LCD Driver
```